Carcassonne

Projet S6

Hadhami Ben Abdeljelil, Arthur Jolivet, Marie Ostertag, Baptiste Roseau

ENSEIRB-MATMECA

13 avril 2018

Encadrant : Steve NGUYEN

Table des matières

1 Présentation du problème

1.1 Règles

L'objectif de ce projet est d'implémenter le jeu Carcassonne afin d'y faire jouer des joueurs en nombre quelconque. Cette première partie permet de rappeler les règles de ce jeu de société.

Cartes

La boîte de jeu est constituée de 72 cartes de paysage de 23 sortes différentes, représentant des types de zones différentes : chaque carte peut être découpée en neuf zones qui représentent soit un chemin, soit un carrefour, soit une abbaye, soit un quartier de ville, soit un pré.

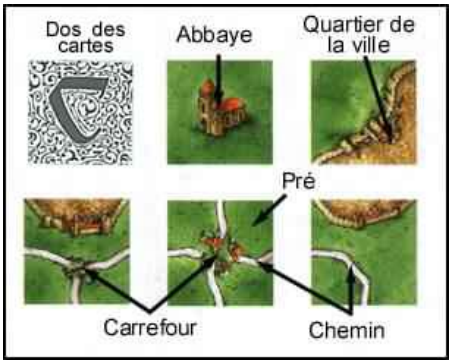


FIGURE 1 – Visuel de certaines cartes

Déroulement

L'objectif de chaque joueur est de marquer un maximum de points en positionnant sur les cartes ses partisans qui sont des petites figurines en bois. Le vainqueur est celui qui a le plus de points à la fin de la partie. En tout début de partie, la carte de départ (toujours la même) est placée sur la table.
Une pile des cartes est positionnée sur la table, les joueurs jouent les uns après les autres jusqu'à ce qu'il n'y ait plus de carte dans la pioche. Quand un joueur a pioché, il doit poser la carte piochée à une position autorisée sur le plateau, c'est-à-dire que les bordures de cette carte doivent coïncider avec celles des cartes adjacentes. Puis il a le droit de mettre un de ses partisans sur un chemin, dans une ville, un pré ou une abbaye sur la carte qu'il vient de positionner.

Comptage des points

Si à la fin de son tour un joueur a un de ses partisans dans une ville achevée (i.e clôturée par des remparts et sans espace vide), ou sur un chemin clôturé, il récupère son partisan marque les points associés et marque les points associés, à savoir 2 points par carte où se trouve la ville ou le chemin, plus les éventuels blasons à raison de deux points par blason.
Enfin, il y a un comptage des points à la fin de la partie, chaque joueur marque 1 point par carte où il a possède un chemin ou une abbaye, plus éventuellement 2 points par carte avec blason. Et pour chaque pré, le ou les joueurs qui possèdent le plus de partisans sont déclarés propriétaires du champ et marquent un point par carte où s'étend le champ.

Pour de plus amples renseignements à propos des règles du jeu Carcassonne, se référer à la règle du jeu.

1.2 Contraintes

Le langage C est imposé pour implémenter notre projet, donc cela implique notamment d'avoir à gérer l'allocation dynamique de la mémoire, mais il offre néanmoins la possibilité de réutiliser nos connaissances autour des types abstraits de données.

Une particularité imposée est de décomposer le projet en deux parties distinctes : les clients et le serveur de jeu. L'intérêt de cette représentation est de créer des clients qu'il est possible d'interchanger, chaque client suivant une stratégie de jeu qui lui est propre. Ainsi, la première étape consiste à transformer chaque fichier client en une librairie dynamique, qui sera passée en paramètre à un serveur de jeu.

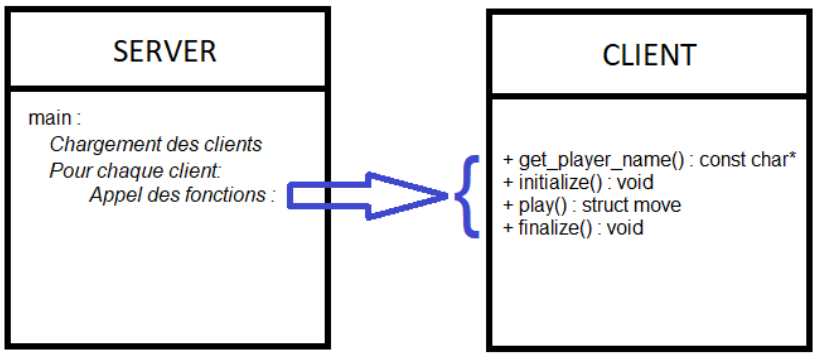


FIGURE 2 – Architecture client-serveur

2 Présentation des types abstraits de données (TADs)

Choix de représentation

Ces règles du jeu nous ont amenés à nous intéresser aux différents points suivants : comment représenter une carte ? Comment représenter le plateau ? Comment représenter un emplacement d'un partisan ? Comment compter les points ?

Le pseudo-algorithme global que nous allons suivre est le suivant :

```
moves = empty_list()
For each player p
   p->initialize(p_id, num_players)
While (the drawing pile is not empty)
   c = draw_card()
   p = compute_next_player()
   m = p->play(c, last_n(moves))
   enqueue(m, moves)
For each player p
   p->finalize()
```

Afin d'obtenir un code aussi simple, plusieurs TADs sont nécessaires. Nous avons choisi d'implémenter une pile pour la pioche, une file pour stocker les actions et les joueurs, et enfin une structure permettant de stocker de façon ordonnée les cartes sur le plateau de jeu, TAD nommé set. Ce dernier sera celui développé en PG116 pour éviter d'avoir à repartir de zéro.
De façon à pouvoir être utilisés à des endroits différents, ces TADs seront polymorphes, ils prendront donc des *void** en argument.
On utilisera toujours un tableau de taille variable, en multipliant sa capacité par 2 s'il y a dépassement, afin de ne pas être bloqué. À l'initialisation, l'utilisateur devra fournir des fonctions de copie, de suppression et d'affichage adaptées aux objets qu'il souhaite manipuler. Pour le TAD set, il devra aussi fournir une fonction de comparaison.

Raisons de ces choix

La pioche a juste besoin d'être parcourue une seule fois, sans nécessité de remettre en jeu une carte, même si la carte piochée n'est pas jouable (auquel cas elle est perdue). Ainsi la pile est une bonne solution, qui sert de point d'arrêt à la boucle principale, quand la pile est vide.

Les joueurs jouent tour à tour, en piochant une carte au début de leur tour, et la partie s'arrête quand la pioche est vide. Les joueurs ont plusieurs cartes à jouer dans une partie, c'est donc plus logique de les représenter par une file.

Nous avons fait en sorte que les emplacements vides adjacents à une carte du plateau- c'est-à-dire qui sont des potentiels lieux pour poser une carte- soient stockés à tout moment. Cette gestion des cartes implique une nouvelle structure, pour les retrouver plus facilement, c'est la structure set implémentée.

2.1 Pile

Principe de fonctionnement

La pile contient un tableau *array*, sa capacité *capacity* et un indice de tête de pile *head*.

```
1 struct stack
2 {
3     void **array;
4     unsigned int capacity;
5     unsigned int head;
6     void* (*operator_copy) (void*);
7     void (*operator_delete) (void*);
8     void (*operator_debug) (void*);
9 };
```

À chaque empilement, on copie l'élément à empiler, puis on le place sur la case de *array* correspondant à *head*, que l'on incrémente par la suite.
Pour ce qui est du dépilement, on crée une copie de d'élément situé dans *array* à l'indice $head-1$, puis on supprime l'élément dans *array* avant de décrémenter *head* et renvoyer la copie.

2.2 File

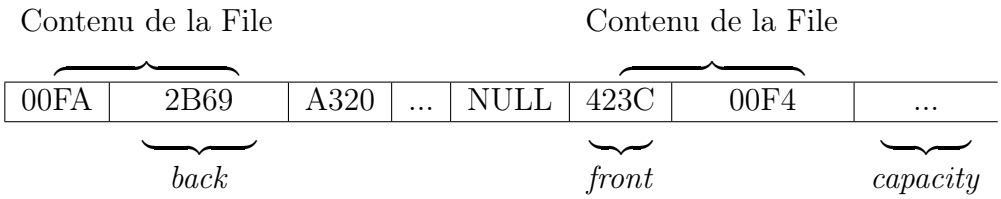
Principe de fonctionnement

La file contient un tableau *array*, sa capacité *capacity*, un indice de tête de file *front* et un indice de fin de file *back*.

```
1  struct queue
2  {
3      void **array;
4      size_t capacity;
5      size_t front;
6      size_t back;
7      void* (*operator_copy) (void*);
8      void (*operator_delete)(void *);
9      void (*operator_debug)(void*);
10 };
```

À chaque enfilement, on copie l'élément à enfiler, puis on le place sur la case de *array* correspondant à *back*. On incrémente ensuite *back* *capacity* de façon à optimiser l'espace disponible dans *array*.

Pour ce qui est du défilement, on peut accéder à l'élément en tête de file via la fonction *queue front* qui renvoie une copie de l'élément situé dans *array* à l'indice *front*.
Pour défiler, on supprime l'élément dans *array front* à l'indice avant d'incrémenter *back* modulo *capacity*.



2.3 Set

Principe de fonctionnement

Le set contient un tableau trié *s*, sa capacité *capacity* et sa taille *set*. Pour le tri du tableau, l'utilisateur doit fournir une fonction de comparaison renvoyant :

— $+1$ pour à $>$
— 0 pour à $=$
— -1 pour à $<$

```
1 struct set
2 {
3     void **s;
4     size_t capacity;
5     size_t size;
6     int (*cmp) (void* x, void* y);
7     void* (*copy) (void* x);
8     void (*delete) (void*);
9 };
```

Pour conserver le tri, on ajoute un élément là où on le recherche. Plus précisément, à chaque ajout, on recherche l'élément à ajouter dans le tableau via une recherche dichotomique, puis, si il n'est pas déjà présent, on en place une copie dans *s* de façon à conserver le tri.
Pour ce qui est du retrait, on recherche l'élément du tableau, puis on le supprime si on le trouve.
Chaque ajout ou retrait se fait en déplaçant les éléments à sa droite d'un indice vers la droite pour un ajout, et d'un indice vers la gauche pour un retrait.

3 Présentation des structures de données

Un certain nombre de structures de données a été implémenté :
- La structure board représentant le plateau de jeu.
- La structure card permettant de décrire une tuile en la découpant en plusieurs sections et de définir son orientation.
- Les structures utilisés pour le comptage des points.

3.1 Structure de données board

Le plateau de jeu est représenté par la structure suivante :

```
1 struct board
2 {
3     struct card* first_card;
4     struct set *cards_set;
5     struct set *meeples_set;
6 };
```

Le champ *first_card* stocke la première carte déposée sur le plateau, *cards_set* stocke la liste des cartes du plateau et *meeples_set*, le nombre de meeples (ou partisans) actuellement sur le plateau.

Les fonctions permettant l'initialisation du plateau de jeu, de positionner une carte, de libérer l'espace mémoire ainsi que toute les fonctions relatives au plateau de jeu n'ont pas encore été implémentées.

3.2 Structure de données card

Une carte est représentée par la structure suivante :

```
1 struct card
2 {
3   struct card_type type;
4   struct card * neighbors[DIRECTION_NUMBER]; // Indexed by enum direction
5   enum orientation orientation;
6 };
```

Avec struct card_type la structure :

```
1 struct card_type
2 {
3   enum card_id id;
4   enum area_type areas[MAX_ZONES]; // indexed by enum place
5 };
```

areas est un tableau de taille 13 (correspondant au découpage en 13 zones de chaque carte) de type enum aera_type. Une zone est soit un FIELD, une CITY, un ROAD, une ABBEY, ou une INTERSECTION.
- Le champ *neighbors* de struct card est un tableau contenant des pointeurs vers quatre cartes entourant la carte en question.

— *orientation* détermine l'orientation de la carte par rapport à card_type (même orientation, rotation de 45 °, rotation de 90 ° ou rotation de 180 °).

Un certain nombre de fonctions relatives à cette structure de données ont été implémentée :

— *card__id_to_type* permet, à partir d'une enum card_id de fournir la structure card_type correspondante.
— *card_init* permet de fournir une structure card à partir d'une enum card_id.
— *card_free* libère la mémoire réservée à la structure card.
— *card__get_area* fournit, à partir d'une enum place et d'une carte l'enum area type de cette zone.
— *card__are_matching_direction* décide si deux cartes coïncident suivant une direction donnée.
— *card__link_at_direction* permet de relier deux cartes entre elles selon une certaine direction.
— *card__set_orientation* modifit l'orientation d'une carte.

3.3 Structure de données pour le comptage des points

Le nombre de points de chaque joueur sera stockée dans la structure **Player** par un entier non signé : *score*.
A la fin de chaque tour d'un joueur, le serveur exécute un parcours de graphe à partir de la dernière carte posée afin de déterminer si celle-ci a fermée des zones, et dans ce cas si des partisans étaient placés sur la zone fermée. Le parcours calcule pour chaque joueur disposant d'un partisan sur la zone fermée son score.

4 Fichiers annexes

Représentation des cartes

TAD pile, utilisé pour gérer la pioche des cartes

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "stack.h"
#include "utils.h"

#define DEFAULT_STACK_CAPACITY 2


int my_random(int min, int max){ //min included, max exlcuded
  return min + (rand() % (max -  min));
}

////////////////////////////////////////////////////////////////////
///     STACK STRUCTURE
////////////////////////////////////////////////////////////////////

struct stack
{
    void* *array;
    unsigned int capacity;
    unsigned int head;
    void* (*operator_copy) (void*);
    void (*operator_delete) (void*);
    void (*operator_debug) (void*);
};


////////////////////////////////////////////////////////////////////
///     STACK FUNCTIONS IMPLEMENTATION
////////////////////////////////////////////////////////////////////

struct stack *stack__empty(void* copy_op, void* delete_op, void* debug_op)
{
    struct stack *s = malloc(sizeof(struct stack));
    if (s == NULL) {
        exit_on_error("Malloc failure on: struct stack*");
    } else {
        s->capacity = DEFAULT_STACK_CAPACITY;
        s->head = 0;
        s->array = malloc(sizeof(void*) * (s->capacity));
        s->operator_copy = copy_op;
        s->operator_delete = delete_op;
        s->operator_debug = debug_op;
    }
    return s;
}

```

```
49 int stack__is_empty(struct stack *s)
50 {
51     if (s == NULL)
52         return -1;
53 
54     return (s->head == 0);
55 }
56 
57 int stack__push(struct stack *s, void* element)
58 {
59     if (s == NULL || s->array == NULL || element == NULL)
60         return -1;
61 
62     /* Adjust capacity if necessary */
63     if (s->head == s->capacity){
64         s->capacity = s->capacity * 2;
65         s->array = realloc(s->array, sizeof(void*) * s->capacity);
66         if (s->array == NULL)
67             return -1;
68     }
69 
70     s->array[s->head] = s->operator_copy(element);
71     s->head++;
72     return 0;
73 }
74 
75 void* stack__peek(struct stack *s)
76 {
77     if (s == NULL || s->array == NULL || stack__is_empty(s))
78         return NULL;
79 
80     return s->operator_copy(s->array[s->head-1]);
81 }
82 
83 void* stack__pop(struct stack *s)
84 {
85     if (s == NULL || s->array == NULL || stack__is_empty(s))
86         return NULL;
87 
88     // Adjust capacity if necessary
89     if ((s->head <= s->capacity / 4) && (s->capacity > DEFAULT_STACK_CAPACITY))
       {
90         s->capacity = s->capacity / 2;
91         s->array = realloc(s->array, sizeof(void*) * s->capacity);
92         if (s->array == NULL)
93             return NULL;
94     }
95 
96     s->head--;
97     void* returned = s->operator_copy(s->array[s->head]);
98     s->operator_delete(s->array[s->head]);
99     s->array[s->head] = NULL;
100    return returned;
101 }
102 
```

```
103 size_t stack__length(struct stack *s)
104 {
105     if (s == NULL)
106         return 0;
107 
108     return s->head;
109 }
110 
111 void stack__mix(struct stack *s)
112 {
113     if (s == NULL || s->array == NULL)
114         return;
115 
116     size_t a;
117     size_t b;
118     void* tmp;
119     for (unsigned int i = 0; i < s->head; i++){
120         a = my_random(0, s->head);
121         b = my_random(0, s->head);
122         tmp = s->array[a];
123         s->array[a] = s->array[b];
124         s->array[b] = tmp;
125     }
126 }
127 
128 void stack__free(struct stack *s)
129 {
130     if (s == NULL || s->array == NULL)
131         return;
132 
133     for (unsigned int i = 0; i < s->head; i++){
134         if (s->array[i] != NULL)
135             s->operator_delete(s->array[i]);
136     }
137     free(s->array);
138     free(s);
139 }
140 
141 void stack__debug(struct stack *s)
142 {
143     if (s == NULL || s->array == NULL)
144         return;
145 
146     if (stack__is_empty(s) != 0)
147         printf("Stack is empty.");
148     else
149         for (unsigned int i = 0; i < s->head; i++)
150             s->operator_debug(s->array[i]);
151 }
```

TAD file, utilisé pour sauvegarder les dernières actions

```
1 #include <stdio.h>
```

```c
#include <stdlib.h>
#include <assert.h>
#include "queue.h"
#include "utils.h"

#define DEFAULT_QUEUE_CAPACITY 2 //Must be a power of 2

////////////////////////////////////////////////////////////////////
///     QUEUE STRUCTURE
////////////////////////////////////////////////////////////////////

struct queue
{
    void **array;
    size_t capacity;
    size_t front;
    size_t back;
    void* (*operator_copy) (void*);
    void (*operator_delete)(void *);
    void (*operator_debug)(void *);
};


////////////////////////////////////////////////////////////////////
///     QUEUE FUNCTIONS IMPLEMENTATION
////////////////////////////////////////////////////////////////////

struct queue *queue__empty(void* copy_op, void* delete_op, void* debug_op)
{
    struct queue *q = malloc(sizeof(struct queue));
    if (q == NULL) {
        exit_on_error("Malloc failure on: struct queue*");
    } else {
        q->array = malloc(sizeof(void *) * DEFAULT_QUEUE_CAPACITY);
        if (q->array == NULL)
            exit_on_error("Malloc failure on: void**");
        else
            for (size_t i = 0; i < DEFAULT_QUEUE_CAPACITY; i++)
                q->array[i] = NULL;

        q->capacity = DEFAULT_QUEUE_CAPACITY;
        q->front = 0;
        q->back = 0;
        q->operator_copy = copy_op;
        q->operator_delete = delete_op;
        q->operator_debug = debug_op;
    }

    return q;
}

int queue__is_empty(struct queue *q)
{
    if (q == NULL)
        return 1;
```

```
57
58      return queue__length(q) == 0;
59  }
60
61  int queue__enqueue(struct queue *q, void* element)
62  {
63      if (q == NULL || element == NULL)
64          return -1;
65
66      // Adjust capacity if necessary
67      if (queue__length(q) == q->capacity) {
68          size_t prev_capacity = q->capacity;
69          q->capacity *= 2;
70          q->array = realloc(q->array, sizeof(void *) * q->capacity);
71          if (q->array == NULL)
72              exit_on_error("Realloc failure on: void**");
73          else
74              for (size_t i = prev_capacity; i < q->capacity; i++)
75                  q->array[i] = NULL;
76
77          q->back = prev_capacity;
78          for (size_t i = 0; i < q->front; i++) {
79              q->array[q->back] = q->array[i];
80              q->array[0] = NULL;
81              q->back++;
82          }
83      }
84
85      q->array[q->back] = q->operator_copy(element);
86      q->back = (q->back + 1) % q->capacity;
87
88      return 0;
89  }
90
91  int queue__dequeue(struct queue *q)
92  {
93      if (q == NULL || queue__is_empty(q))
94          return -1;
95
96      q->operator_delete(q->array[q->front]);
97      q->array[q->front] = NULL;
98      q->front = (q->front + 1) %q->capacity;
99
100     if (queue__length(q) < q->capacity / 2 && q->capacity > 2) {
101         size_t new_pos = 0;
102         for (size_t i = q->front; i != q->back; i = (i+1) % q->capacity ) {
103             q->array[new_pos] = q->array[i];
104             q->array[i] = NULL;
105             new_pos++;
106         }
107         q->front = 0;
108         q->back = positive_modulo((int) (new_pos), (int) q->capacity);
109
110         q->array = realloc(q->array, sizeof(void *) * q->capacity / 2);
111         if (q->array == NULL)
```

```
112             exit_on_error("Realloc failure on: void**");
113         else
114             q->capacity /= 2;
115     }
116 
117     return 0;
118 }
119 
120 void* queue__front(struct queue *q)
121 {
122     if (q == NULL || queue__is_empty(q))
123         return NULL;
124 
125     return q->operator_copy(q->array[q->front]);
126 }
127 
128 void* queue__back(struct queue *q)
129 {
130     if (q == NULL || queue__is_empty(q))
131         return NULL;
132 
133     if (q->back == 0)
134         return q->operator_copy(q->array[q->capacity - 1]);
135     else
136         return q->operator_copy(q->array[q->back-1]);
137 }
138 
139 size_t queue__length(struct queue *q)
140 {
141     if (q == NULL)
142         return 0;
143 
144     if (q->back > q->front || q->array[q->front] == NULL)
145         return q->back - q->front;
146     else
147         return q->back + q->capacity - q->front;
148 }
149 
150 void queue__free(struct queue *q)
151 {
152     if (q == NULL)
153         return;
154 
155     for (size_t i = 0; i < q->capacity; i++)
156         if (q->array[i] != NULL)
157             q->operator_delete(q->array[i]);
158 
159     free(q->array);
160     free(q);
161 }
162 
163 void queue__debug(struct queue *q)
164 {
165     if (q == NULL)
166         return;
```

```
167
168     printf("=== QUEUE DETAILS ===\n");
169     printf("Queue capacity: %zu\n", q->capacity);
170     printf("Queue size: %zu\n", queue__length(q));
171
172     printf("Queue content in queue order...\t");
173     for (size_t i = q->front; i != q->back; i = (i+1) % q->capacity ) {
174         q->operator_debug(q->array[i]);
175     }
176     printf("\nQueue content in array order...\t");
177     for (size_t i = 0; i < q->capacity; i++) {
178         q->operator_debug(q->array[i]);
179     }
180     printf("\n");
181 }
```

TAD set, pour stocker les positions possibles pour poser la prochaine carte

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <assert.h>
4  #include "set.h"
5
6  #define BASIC_SET_SIZE 8 //must be a power of two
7
8  struct set
9  {
10     void **s;
11     size_t capacity;
12     size_t size;
13     int (*cmp) (void* x, void* y);
14     void* (*copy) (void* x);
15     void (*delete) (void*);
16 };
17
18
19 //dichotomic search
20 size_t find(struct set const *set, size_t beg, size_t end, void* x)
21 {
22     assert(x != NULL);
23     if (beg >= end)
24         return beg;
25
26     size_t m = (beg + end)/2;
27     if (set->s[m] == x)
28         return m;
29     if (set->cmp(x, set->s[m]) == 1)
30         return find(set, m + 1, end, x);
31     else
32         return find(set, beg, m, x);
33 }
34
```

```
35
36 void shift_right(struct set const *set, size_t begin, size_t end)
37 {
38     if ((end >= set->capacity - 1) || (end > set->size))
39         return;
40     //Shift from [begin, end] to [begin + 1, end + 1] by a right to left path
41     size_t k = end + 1;
42     while (k > begin) {
43         set->s[k]=set->s[k-1];
44         k--;
45     }
46 }
47
48
49 void shift_left(struct set const *set, size_t begin, size_t end)
50 {
51     if ((begin == 0) || (end != set->size - 1))
52         return;
53
54     //Shift from [begin, end] to [begin - 1, end - 1] by a left to right path
55     size_t k = begin - 1;
56     while (k < end) {
57         set->s[k]=set->s[k+1];
58         k++;
59     }
60 }
61
62
63
64 struct set *set__empty(void* (*copy) ( void* x),
65                        void (*delete) (void*),
66                        int (*compare) ( void* x,  void* y))
67 {
68     struct set *set = malloc(sizeof(struct set));
69     set->capacity = BASIC_SET_SIZE;
70     set->s = malloc(sizeof(void*)*set->capacity);
71     set->size = 0;
72     set->copy =copy;
73     set->delete = delete;
74     set->cmp = compare;
75
76     return set;
77 }
78
79
80 int set__is_empty(struct set const *set)
81 {
82     return (set->size == 0);
83 }
84
85
86 int set__add(struct set *set, void* x)
87 {
88     if  (x == NULL)
89         return 1;
```

```
90
91      size_t pos = find(set, 0, set->size, x);
92      if  ((pos < set->size) && (set->cmp(set->s[pos], x) == 0))
93          return 1;
94
95      //Augment memory if  needed
96      if (set->size == set->capacity) {
97          set->capacity = set->capacity * 2;
98          set->s = realloc(set->s, sizeof(void*)*set->capacity);
99          assert(set->s != NULL);
100     }
101
102     //Add x into the set
103     shift_right(set, pos, set->size);
104     set->s[pos] = set->copy(x);
105     set->size++;
106
107     return 0;
108 }
109
110
111 int set__remove(struct set *set, void* x)
112 {
113     assert(set->s != NULL);
114     if  (x == NULL)
115         return 1;
116
117     size_t pos = find(set, 0, set->size, x);
118     if  ((set->size == 0)
119             || (pos >= set->size)
120             || (x == NULL)
121             || ((pos < set->size) && (set->cmp(set->s[pos], x) != 0)))
122         return 1;
123
124     //Remove the element
125     set->delete(set->s[pos]);
126     shift_left(set, pos + 1, set->size - 1);
127     set->size--;
128
129     //Remove some allocated memory if needed
130     if (set->size < set->capacity / 4) {
131         set->capacity = set->capacity / 4;
132         set->s = realloc(set->s, sizeof(void*)*set->capacity);
133         assert(set->s != NULL);
134     }
135
136     return 0;
137 }
138
139
140 int set__find(struct set const *set, void* x)
141 {
142     size_t pos = find(set, 0, set->size, x);
143     return  ((pos < set->size) && (set->cmp(set->s[pos], x) == 0));
144 }
```

```
145
146
147 size_t set__size(struct set const * set)
148 {
149     return set->size;
150 }
151
152
153 void set__free(struct set *set)
154 {
155     for (size_t i = 0; i < set->size; i++)
156         set->delete(set->s[i]);
157     free(set->s);
158     free(set);
159 }
160
161
162 struct set *set__filter(const struct set *set, int (*filter) (const void*))
163 {
164     struct set *filtered_set = set__empty(set->copy, set->delete, set->cmp);
165     size_t i = 0;
166     while ((i < set->capacity) && (i < set->size)) {
167         //Augment memory if  needed
168         if (filtered_set->size == filtered_set->capacity) {
169             filtered_set->capacity = filtered_set->capacity * 2;
170             filtered_set->s = realloc(filtered_set->s, sizeof(void*)*
    filtered_set->capacity);
171             assert(filtered_set->s != NULL);
172         }
173         if (filter(set->s[i])) {
174             filtered_set->s[filtered_set->size -1] = set->copy(set->s[i]);
175             filtered_set->size++;
176         }
177         i++;
178     }
179     return filtered_set;
180 }
181
182
183 void set__debug_data(const struct set *set, void (*print_data) (const void*))
184 {
185     size_t i = 0;
186     if (set__is_empty(set)) {
187         printf("Empty set ");
188         return;
189     }
190     while(i < set->size) {
191         print_data(set->s[i]);
192         i++;
193     }
194 }
```